STUDENT NAME: ______________________ **STUDENT ID:** ____________________

CS122C/222 – Fall 2018, Midterm Exam

Principles of Data Management

Department of Computer Science, UC Irvine

Prof. Chen Li

(Max. Points: 100)

Instructions:

- **This exam has six (6) questions.**
- **This exam is closed book. However, you can use one cheat sheet (A4 size).**
- **The total time for the exam is 80 minutes, so budget your time accordingly.**
- **Be sure to answer each part of each question after reading the whole question carefully.**
- **If you don't understand something, ask for clarification.**
- **If you still find ambiguities in a question, write down the interpretation you are taking and then answer the question based on that interpretation.**

QUESTION	POINTS	SCORE
1	20	
2	24	
3	20	
4	20	
5	16	
TOTAL	100	

Question 1 Short questions (20 points)

a) (4 pts) Explain how the buffer manager implements the PIN/UNPIN operations on a frame in the buffer pool to make sure the frame can be replaced properly.

- The buffer manager keeps a reference count for a frame;
- When a caller calls “PIN”, the buffer manager increments the count by 1;
- When a caller calls “UNPIN”, the buffer manager decrements the count by 1;
- During the cache replacement, the buffer manager replaces the frame only if its reference count is 0, which means that no callers are still using this frame.

b) (4 pts) Some databases allow users to create a B+ tree index on one attribute, while the index stores the value of another attribute, even though the second attribute is not used in the total order of the index keys. For instance, consider a table **Product(id, name, price, category)**. The database can allow us to create a B+ tree on the “category” attribute, while we can still store “price” values in this index, even though it is NOT a composite index. Explain the main benefit of creating such an index.

By adding the ”price” attribute value to the B+ index of “category”, we can use this index to do an index-only plan for a query that needs to access the two values of each record. For instance:

```
SELECT category, SUM(price)
FROM Product
GROUP BY category;
```

This query can be answered by this index using an index-only plan.

c) (4 pts) Briefly explain the three portions of the time in each disk IO (read operation).

- Seek time: move the disk head to the track of the page to read;
- Rotational delay: spin the spindle until the head is at the beginning of the page;
- Transfer time: the time to read the page form the disk.

d) (4 pts) Explain two main differences between OLTP queries and OLAP queries.

- OLTP queries: small queries, access a few records, benefit from indexes, need fast response time, suitable for row stores.
- OLAP queries: big queries, access many records, tend to access a few columns, indexes not very useful, fewer queries, take longer time, suitable for column stores.

e) (4 pts) In Project 2, briefly explain how tombstones are used in the function **`updateRecord()`**.

When updating a record, if the new record size is bigger, and the current page doesn't have enough space for the new record, we insert the new record to a different page, and add a tombstone (RID of the new record) to the place of the original record.

Question 2: Record Manager (24 points)

Suppose we have a table with the following schema:

Tweet (*id* **INT**, *user* **VARCHAR(**30**)**, *msg* **VARCHAR(**30**)**, *lat* **FLOAT**, *long* **FLOAT**, *retweetCount* **INT)**

We store the table as a heap file of variable-length records. Each record is stored as a sequence of bytes with a directory of pointers. Assume that:

1. The directory uses a 2-byte offset to store the ending position for each field.
2. The offset starts from 0, which is the beginning of the directory.
3. The schema is known in all record operations so there is no need to store the number of fields in each record.
4. Integers and floating-point numbers are 4 bytes each.
5. Each NULL value is represented using a special value -1 in the corresponding pointer in the directory.

Consider the following record:

(4321, “Bob”, “hi”, NULL, NULL, 5)□.

a) (4 pts) Fill in the following byte array using the format stated above. Clearly indicate the number of bytes in each segment and all their values.
The offset value could be the end of the field or the beginning of following field, and we take both as correct as long as your solution is consistent.

15		18		20		-1		-1		24		4321				B	o	b	h	i	5									
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30

or

16		19		21		-1		-1		25		4321				B	o	b	h	i	5									
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30

b) (5 pts) Fill in the following byte array using the **“*data”** format defined in the “`insertRecord()`” function of RBFM in our course project, as following:

[null-indicator] [value of 1st field] [value of 2nd field] …

Each Int/Real uses 4 bytes and each Varchar uses 4 bytes for the length followed by actual values. For the null indicator, show the values of the 0/1 bits. Clearly show the number of bytes in each segment and all their values.

00011000	4321	3	B	o	b	2	h	i	5	▨
0	1 2 3 4	5 6 7 8	9	10	11	12 13 14 15	16	17	18 19 20 21	22 23 24 25 26 27 28 29 30

c) Suppose we store records with each page size of 4,096 bytes as shown in the following diagrams. Assume each offset takes 2 bytes, and each array length also uses 2 bytes. Slots are inserted from right to left in ascending RID numbers. Suppose initially the page is empty. Fill the missing values in the following diagrams after those operations are done, including the record offsets and lengths, number of bytes in free space, and number of slots.

c-1) (5 pts) Insert 3 records, R1, R2, and R3, with byte array sizes of 30, 25, and 35, respectively.

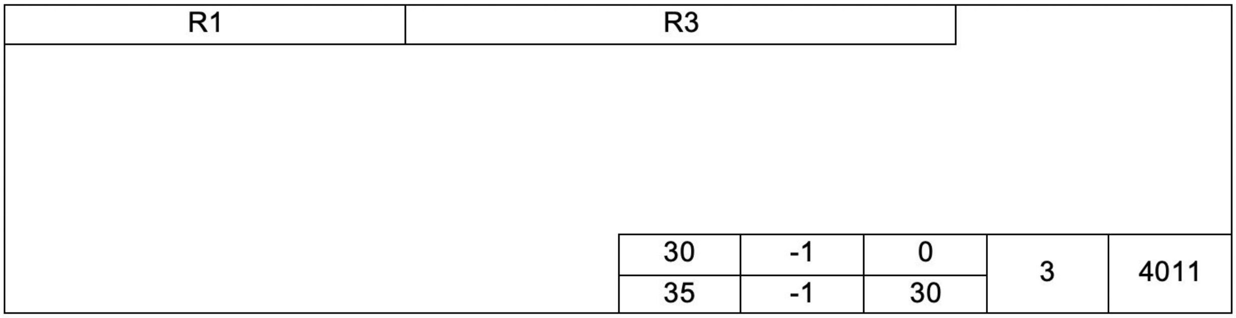
R1	R2	R3

55	30	0	3	3986
35	25	30		

Freespace = 4096 – (30 +25 + 35) – (4 + 4 + 3*4) = 3986

In the following two subquestions, in addition to showing the calculations, make sure to explain the main idea of your solution. This explanation may give you partial credit in case you made a mistake in your earlier solution.

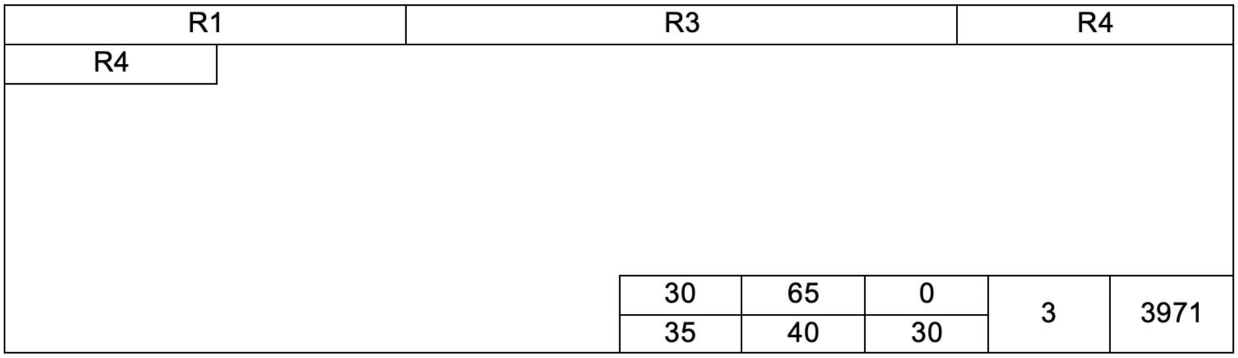
c-2) (5 pts) Delete record R2 (of size 25). Assume the page is compacted immediately.



30	-1	0	3	4011
35	-1	30		

Freespace = 3986 + 25 = 4011

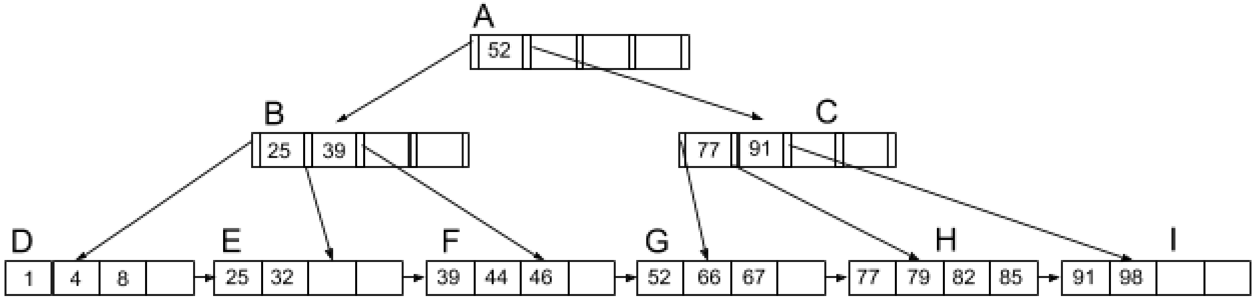
c-3) (5 pts) Insert record R4 with an array of 40 bytes. Assume a deleted rid will be reused immediately in the next insertion.



30	65	0	3	3971
35	40	30		

Freespace = 4011 - 40 = 3971

Question 3: B+ tree (20 points)
Consider the following unclustered B+ tree index on the “price” field of a relation **Products(itemID int, price int)**.



a) (5 pts) For the following query, assume each record in the answer resides in a different page. Calculate the number of disk I/Os if we use this index to answer the following query. Briefly explain your calculation.

SELECT * FROM R WHERE PRICE ≥ 39 AND PRICE ≤ 78;

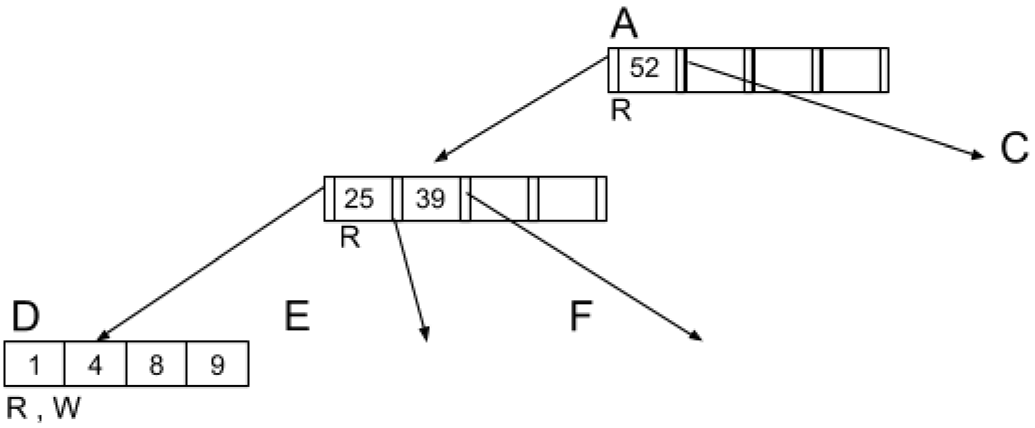
To answer the query we need a total of 12 I/Os

- 5 index page I/Os: A -> B -> F -> G -> H
- 7 file page I/Os to fetch records: 39, 44, 46, 52, 66, 67, 77

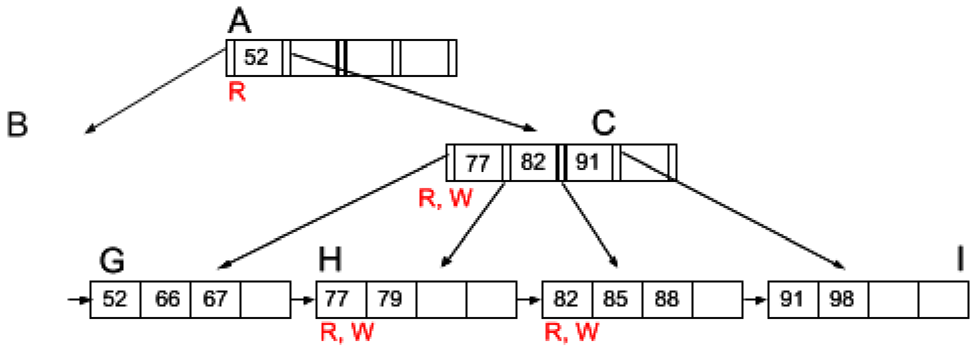
For the following questions, **draw the updated B+ tree after each operation**. If there is no ambiguity, you can just draw the part that is changed. For each operation, clearly add an “R” (for read), “W” (for write), and “A” (meaning appending a page, which includes its “Write” operation), on all the affected pages, as well as the original A/B/C/.../I labels.

Assume:
(1) Each node can hold up to 4 key entries.
(2) For delete operations, you can borrow from the **right** sibling (one node only).
(3) For insert operations, you need to do split, not rotation.

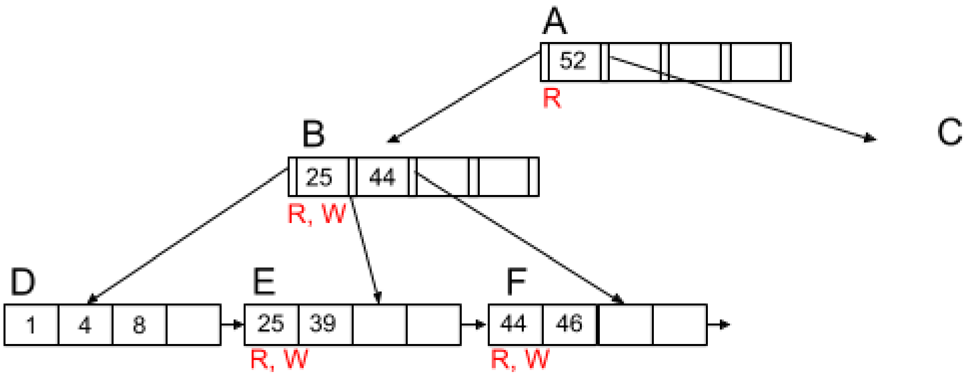
The following is an example after inserting 9 on the **original** tree:



b) (5 pts) Insert 88 on the **original** tree.



c) (5 pts) Delete 32 from the **original** tree.



d) (5 pts) In the lecture we have discussed the idea of bulk loading in B+ tree for a large dataset. List two main advantages of this approach over the approach of doing multiple insertions.

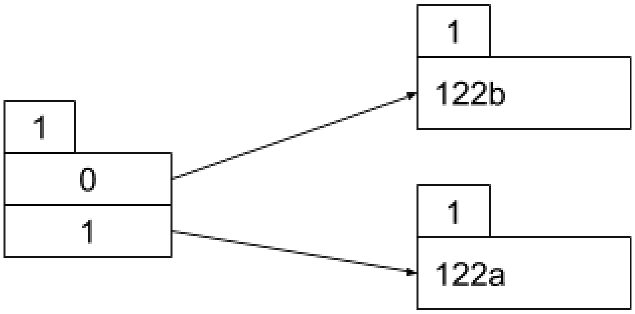
- Has advantages for concurrency control.
- Fewer I/Os during build.
- Leaves will be stored sequentially (and linked).
- Can control “fill factor” on pages.
- Can optimize non-leaf splits more than shown.

Question 4: Hashing (20 points)

Consider storing a set of course codes in a table, which has a dynamic hash table. Suppose we want to insert the following 10 entries. We are using a hash function called ***h,*** which simply computes a hash value of a course code based on its binary value (their hash values are also provided below). Only 2 records can fit into a bucket/page. Assume that the bits of h(k) are used in least-to-most order of significance. For example, for a binary value "0010", we use the two rightmost bits "10" to find the bucket.

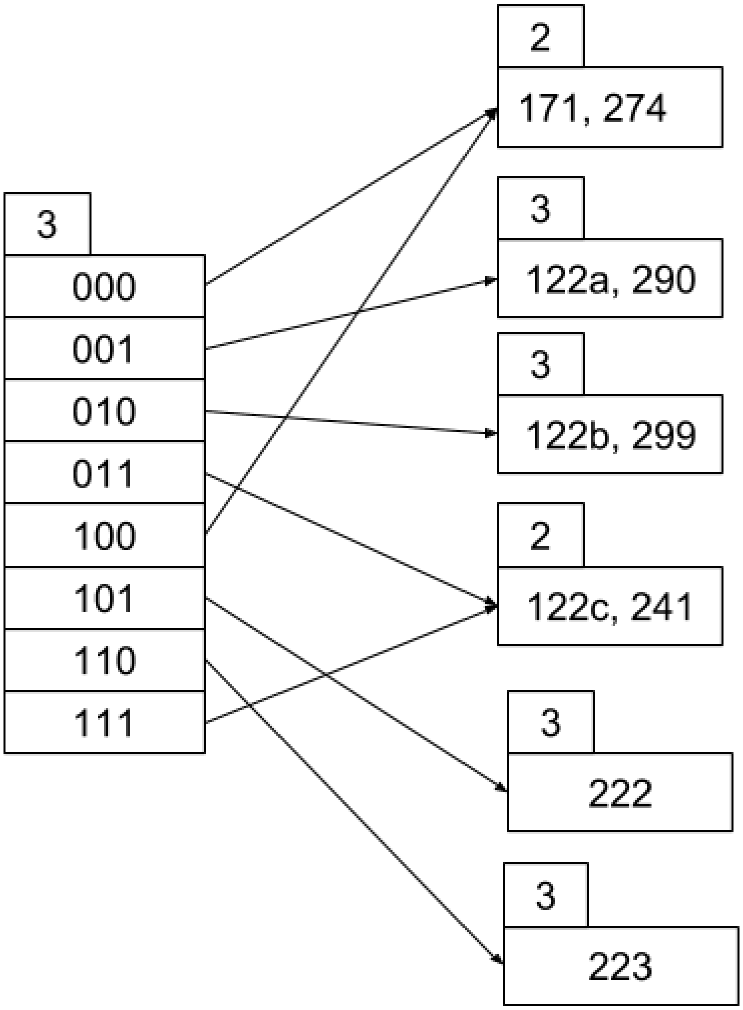
Course Code	Hash Value
122a	0001
122b	0010
122c	0011
171	0100
222	0101
223	0110
241	0111
274	1000
290	1001
299	1010

a) (10 pts) Suppose the index is based on the **extendible hashing** scheme. Below is the state of dynamic hashed index after inserting the first two data entries. Draw a diagram representing the final state of the dynamic hash index after inserting all the 10 given data entries. **You are highly recommended to use a pencil and an eraser to work on your solution.**



*copied for your convenience

Course Code	Hash Value
122a	0001
122b	0010
122c	0011
171	0100
222	0101
223	0110
241	0111
274	1000
290	1001
299	1010



b) (4 pts) In hash tables, give two reasons for the hashed values to be skewed, i.e., they are not evenly distributed across all possible values.

(1) the hash function is poorly designed that results in many conflicts
(2) there are a lot of duplicate key values
Note that range skew (i.e., some ranges contain a large number of values) is not necessarily a problem for a hash index, since hashing will redistribute all values.

c) (6 pts) Suppose in **extendible hashing**, our split policy is the following: whenever an insertion causes a new overflow page, split the pages with this hash value, *only once*.

Also suppose in **linear hashing**, our split policy is the following: whenever an insertion causes a new overflow page, split the page(s) pointed by the "Next" pointer, *only once*.

Under these assumptions, explain how skewed hash values impact the query performance of extendible hashing and linear hashing, respectively?

For extendible hashing, there are no overflow chains. However,skewed hash values can increase the size of the directory, incurring an extra I/O for each query if the directory cannot be cached.

There is no explicit directory for linear hashing. However, skewed hash values can cause long overflow chains to develop, negatively impacting query performance.

Question 5: Indexing Performance (16 points)

Suppose you are in charge of managing the database for Uber. Consider a table

UberRides(<u>rideID</u>, driverID, customerID, origin, destination, fare)

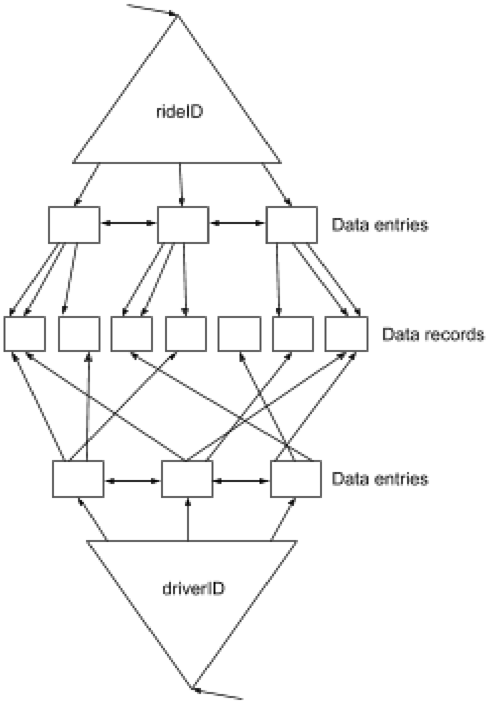
containing 1,000,000,000 records, and each page contains 100 records. We have the following assumptions.

- "rideID" is an integer value distributed uniformly over the range [0 - 999,999,999].
- "driverID" is an integer value distributed uniformly over the range [0 - 999,999].
- "customerID" is an integer value distributed uniformly over the range [0 - 99,999,999].
- "origin" and "destination" are variable-length strings with a not-null constraint and a maximum length of 100 bytes.
- "fare" is a double value distributed over the range [0 - 5,000]

We have the following indexes on the table:

1) Clustered B+ tree index on "rideID".
2) Unclustered B+ tree index on "driverID".
3) Unclustered B+ index on "customerID".
4) Unclustered B+ tree index on composite key "<origin, destination, fare>".

a) (4 pts) Draw a diagram to illustrate the main idea of the clustered index and the "driverID" unclustered index. Show clearly the content of their leaf nodes.



For each of the following queries, decide the most efficient access strategy. Briefly explain the reason.

b) (4 pts) SELECT U.origin, U.destination FROM UberRides U WHERE U.rideID = 45;

We should use index 1.
The predict on rideID is very selective.

c) (4 pts) SELECT count(*) FROM UberRides U WHERE U.customerID > 5,000 and U.customerID < 7000.

We should use index-plan using index 3.
We just need to count the number of RIDs in the leaf nodes that satisfy the predict without accessing the file of records

d) (4 pts) SELECT AVG(U.fare) FROM UberRides U WHERE U.origin = “Irvine”.

We should use index-plan using index 4.
Since the composite key is <origin, destination, fare>, we can easily calculate the fare for a specific predict without returning to the file of records.